/40
/40
/40
/30
/30
/20
/200

CPS 512/590 second midterm exam, 11/3/2015

Your name please: ______________________________________

Part 1. Spanner

These questions pertain to multi-version transactions as used in the Spanner system. Answer each question with a phrase or two or maybe a sentence or two. [40 points]

a) How does Spanner enforce serializability of update (read/write) transactions?

Two-phase locking (**2PL**).
Many answers began discussing timestamps, but these are not pertinent to the question: Spanner uses timestamp ordering only for transactions that are declared read-only before they start.

b) Can two Spanner transactions commit with the same timestamp?

Yes. Since Spanner has no central point of control for transactions, two transactions may commit with the same timestamp if they have no tablets in common, or no Paxos groups in common.

This was not intended as a “trick”. I gave partial credit for No answers that revealed an understanding of how timestamps are assigned.

c) Spanner is a key/value store. Why is it necessary for each Spanner replica group to use a Replicated State Machine (RSM) model, which imposes an order on write requests even if they affect different keys?

A Spanner **tablet is a complex data structure** that stores key/value pairs lexicographically ordered by key, with an index that supports range queries. So updates to the value for one key may affect how others are stored and/or indexed. In the RSM model, operations on the tablet are deterministic and all replicas execute the same requests in the same order, so all replicas reach the same state, even if the application (i.e., the tablet code) implements those operations in a complex way.

d) A Spanner group leader may force a transaction to abort. Why / under what circumstances?

Potential deadlock. This may occur when a transaction attempts to acquire a lock held by another transaction, hen blocking to wait for the lock could create a risk of deadlock per the “wound-wait” strategy.

Part 2. Déjà vu: Chubby revisited

These questions are similar to questions on the first midterm. Answer each question with a few phrases or maybe a sentence or two. [40 points]

a) In Chubby (and in Lab #2), applications acquire and release locks with explicit **acquire()** and **release()** primitives. Leases create the possibility that an application thread holding a lock (the lock holder) loses the lock before releasing it. Under what conditions can this occur?

A **network partition**. That is the only case! (OK, it could also happen if the lock service fails.) Otherwise, the client renews the lease as needed until the application releases the lock. The lock server does not deny a lease renewal request from a client, even if another client is contending for the lock.

b) In this circumstance (a), how does a Chubby lock client determine that the lease has been lost? How does the application determine that the lock has been lost?

The lock client knows the lease is lost if the lease expires, i.e., the client receives no renewal response before the expiration time. The application receives an exception/notification (from the client) that it is in **jeopardy** and (later) that it has lost its session with Chubby.

c) Chubby maintains a single lease for each client's session with Chubby, rather than a lease for each individual lock that a client might hold. What advantages/benefits does this choice offer for Chubby and its clients?

It is more **efficient**: the cost of handling lease renewals is amortized across all locks held by a client. It is also more tolerant to transient network failures and server fail-over delays because Chubby renews a client's session lease automatically on every message received from a client. If the session lease is lost, then the client loses all locks and all other Chubby resources that it holds.

Note: the per-client single session lease does NOT mean that there is now only "one big global lock" or that clients have to implement mutual exclusion themselves. The lock abstraction is unchanged. Some students seem to be stuck on this misconception.

Part 2. Déjà vu: Chubby revisited…

d) Given that Chubby can complete a write after agreement from a subset of replicas, how do the other replicas learn of the write under normal failure-free operation?

Chubby uses a consensus protocol (Paxos) to replicate updates. The primary/leader/coordinator sends (multicasts) the write to all replicas. Although the primary does not wait for all responses, **all replicas receive the initial multicast of the update from the primary/coordinator** if there are no failures.

If there are failures, a recovering replica receives updates/slots/entries that it missed, either from the leader or from a peer (as described for VR).

Part 3. Causality

These questions pertain to the concepts employed in the Bayou system. Please keep your answers tight. [40 points]

a) Give an example of two vector clocks v_1 and v_2 in which $v_1 < v_2$ (v_1 **happened-before** v_2).

For three nodes: v1 = (1, 0, 0), v2 = (1, 1, 0). v2 dominates v1.

Some students drew nice happened-before timelines with no vector clocks. Half credit for that: I really wanted examples of the vector clock values themselves.

b) Give an example of two vector clocks v_1 and v_2 in which v_1 is **concurrent** with v_2.

For three nodes: v1 = (1, 0, 0), v2 = (0, 1, 0). Neither clock dominates the other.

c) Give hypothetical real-world example in which a client learns of updates in an order that does not respect potential causality, i.e., that violates causal ordering. Why is this bad?

You need an example of two updates u1 and u2 with u1 < u2, and a read u3 that operates on a system state that reflects u2 but not u1, i.e., where u2 is applied before u1. **Note**: if u1 and u2 in your example are concurrent, then it is just an ordinary race and reordering them does not violate causality.

So: u1={Alice deposits $100 into empty account J}, u2 = {Bob checks J balance and withdraws $100}, u3={Bank issues an overdraft fee on J}.
Or: u1={Alice removes Bob as friend}, u2={Alice insults Bob to her friends}, u3 = {Bob reads the insults}.

d) Bayou clients may learn of updates in different orders, yet these orders always respect causality. How does Bayou assure this property? I can think of several acceptable one-sentence answers. Please try to answer in one carefully worded sentence!

Replicas transfer/apply updates cumulatively in order, according to the prefix property, and order them by logical clocks. Suppose a replica R generates an update w: before telling any other replica S about w, it first tells S of all updates u that R had already learned of (or generated/accepted) at the time that R generated w. By definition, this sequence includes all causally preceding updates u with u<w. Updates flood transitively through the network in this fashion, preserving causal order at each step. Newly learned updates may be interleaved with updates learned previously, but the ordering is by logical clock (accept-stamp order) and so cannot violate causality, and does not change the relative order of known updates.

Part 4. Consensus

These questions pertain to **consensus** algorithms (Viewstamped Replication (VR), Raft, and Paxos). You may answer with reference to any variant of **consensus**. Please keep your answers tight. [30 points]

a) Where is **consensus** on the CAP triangle? How does **consensus** behave under a network partition?

Consensus is **CP**: it is consistent (sound), but becomes unavailable (i.e., not live) in a network partition or other circumstance in which a majority of nodes are unreachable.

b) A fundamental property of **consensus** is that each view (also called term or ballot) has at most one primary (also called a leader). Is it possible for a view to have no primary? How could this occur?

This could occur in VR if the designated **leader for a given view has failed** or is unreachable. It could happen in Paxos or Raft if **no leader obtained a majority** on a given term or ballot.

c) Unnecessary view changes (or elections/scouts) can prevent **consensus** from making progress. Why? Propose a mechanism to prevent unnecessary view changes in practice.

A replica ignores the previous leader once the replica has advanced to a higher term/ballot/view. So: once a majority of replicas have advanced, the previous round is interrupted and no further progress is possible until the view change is complete and a new leader is initialized and accepted. (This could take some time.)

All practical consensus algorithms use carefully chosen **timeouts** to determine when to initiate a new view. If a replica receives a message from the current leader before the timeout expires, it resets the timer and does not advance to a new view. (Note: this is a **leader lease**.) Spanner uses long leader leases to avoid unnecessary leader changes, and Raft randomizes the timeouts to reduce conflicts between candidate leaders. There is a tradeoff: shorter timeouts reduce failover times when the leader fails, but they may force unnecessary view changes when the network is slow. Longer timeouts reduce unnecessary view changes, but have longer failover times.

Part 5. A little more thought [30 points]

Consider an execution of **consensus** with five replicas, in which a leader/primary proposes (prepares) values for a sequence of log entries (also called op-numbers, slots, or indexes). Consider now what happens over a sequence of views, each with a different primary (we may number them P_1, P_2, P_3,…). Give an example in which a primary proposes a value *v* that overwrites a different value *w* that an earlier primary proposed for the same slot, even though the later primary was aware of *w*, i.e., *w* was accepted by at least one member of its voting majority.

The purpose of this question was to evaluate whether you really understand consensus. It is a difficult question. To construct such an example, you need three views:

- In the first view, P1 proposes a value w, and a minority of nodes (including P1) accept it, so **w does not commit**.
- In the second view, P2 does not learn of w, e.g., because its voting majority includes no node that is aware of w. So P2 proposes a value v for the same slot as w. It is immaterial whether v commits or not.
- In the third view, P3 learns of both w and v. P3 then overwrites w with v on any replicas that had accepted w, and the example is complete.

A few points to note:

1. If w commits under P1, the example fails because it is impossible that P2 does not learn of w: since w committed, a majority of nodes know of w, so the majority that elected P2 must include at least one node that knows of w.
2. If P2 does learn of w, the example fails because P2 proposes w again for the same slot (i.e., P2 affirms w). P2 will **never** propose a conflicting value v for that slot. This is true even if w did not commit under P1. In fact, P2 cannot always tell whether w committed under P1 or not. That is why P2 always affirms w: if w did commit, then it is essential that P2 affirms w, so P2 always affirms w just to be safe. So: the example requires that any node that accepts w under P1 fails under P2, so that P2 does not learn of w.
3. If there are only two views instead of three, the example fails. With two views there are only two cases: P2 either did or did not learn of w. If P2 does not learn of w, the example fails because the question requires that the primary that overwrites w does learn of w. If P2 does learn of w, then we are back to point 2.
4. If P3 does not learn of both v and w, the example fails. If P3 learns of w but not v, it affirms w (v could not have committed). If P3 learns of v but not w, then it affirms v.
5. To ensure that P3 learns of w as well as v, the example requires that a node that accepted w under P1 and then fails under P2 recovers under P3, and informs P3 of w. Moreover, some node that accepted v under P2 must survive under P3, so P3 also learns of v. In this case and this case alone does the primary (P3) deliberately overwrite an older accepted update (w) with a newer one (v). This is guaranteed to be safe because w could not have committed. If w had committed, then P2 would have learned of w (point 1) and would have affirmed it (point 2) instead of proposing v for that slot.

Part 5. A little more thought [30 points]…

In general, I was fairly generous with partial credit here.

Most of the answers constructed examples that were correct but did not include all of the required elements, and then simply trailed off and declared victory. I viewed these as pretty good efforts under time pressure.

But some examples went astray in ways that exposed misconceptions about the algorithm. The most common error is that people seem to be confused about "voting" in **consensus** vs. in 2PC. In **consensus**, acceptors/replicas **never vote no** on a value proposed by the leader. They are **sheep**! They always follow the leader! They do whatever the leader says and accept whatever the leader says. Always, no exceptions.

So there are no examples in which the voting majority of the new primary votes for v over w. The leader (P) never chooses values based on a count of the votes! If P hears about a value w for slot s, it affirms w **unless** it also learns of a later update v for s. In that case, P affirms v and overwrites w. P affirms v even if more of its voters had previously accepted w for s, and not v! P may not know if v committed or not, but it does not matter, because P knows for sure that w did **not** commit! If w had committed, then no previous primary (i.e., P2 in the example) would have proposed a conflicting value (v) for the same slot (s) as w, because that primary must have learned of w and so would also have affirmed w. This is the key step of the correctness proof for **consensus**.

Part 6. Mashing it all together [20 points]

We might think of Bayou's approach as "eventual consensus": like eventual consistency the system always makes progress as long as any replicas are reachable and/or can communicate, and like **consensus**, all replicas eventually agree on the same sequence of updates (presuming that all failures eventually resolve). To achieve these properties, Bayou must make stronger assumptions about the application than **consensus** does. Compare/contrast their assumptions about the application.

Both consensus and Bayou assume that the application has an **operational update API that is deterministic**, so that two replicas that execute the same sequence of operations end up in the same final state. Both maintain a **replicated log for the sequence of update operations**. This is the RSM model.

Bayou further presumes that the application can handle rollback and reordering of updates and that it can detect and resolve conflicting updates. The first need arises because replicas may learn of updates late, and so have to insert those updates into their logs. Applications must enable this reordering and also should have some way to inform the user that some states are "tentative" and subject to rollback and change. The second need arises because the replicas may accept concurrent conflicting updates, which the application must reconcile (like Dynamo).

A Bayou application supplies deterministic procedures to handle these needs, including **dependency checks and merge procedures** to detect and reconcile conflicts for update operations.

These procedures are application-supplied and application-specific. The Bayou replica core does not know or care what these procedures do: it merely upcalls them at the right times, and they must work properly or the application will fail. I got some answers that summarized elements of Bayou's optimistic asynchronous replication model: anti-entropy exchanges, version vectors, and so on. These are properties of the application-independent Bayou core, and they pertain only indirectly to Bayou's assumptions about the application.